Personenverwaltungsmodul

Entwickelt von

Johannes Kunz und

Ron Rieck

Inhaltsverzeichnis

[1. Projektaufgabe 3](#_Toc533063343)

[1.1 Die 3 Schichtenarchitektur 3](#_Toc533063344)

[1.2 Das Design-Pattern 4](#_Toc533063345)

[1.3 Unser Anwendungsproblem 8](#_Toc533063346)

[2. Das Produkt 8](#_Toc533063347)

[2.1 Klassendiagramm inkl. Zuordnung 8](#_Toc533063348)

[2.2 Benutzeroberfläche 11](#_Toc533063349)

[2.2.1 GUI 11](#_Toc533063350)

[2.2.2 TUI 11](#_Toc533063351)

[2.3 Datenhaltung 12](#_Toc533063352)

[2.3.1 ER-Modell 12](#_Toc533063353)

[2.3.2 XML 12](#_Toc533063354)

[2.3.3 SQLite 13](#_Toc533063355)

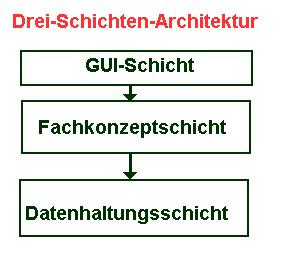
[2.4 Realisierung des Designpatterns 13](#_Toc533063356)

[2.5 Sequenzdiagramm für die Kommunikation der 3 Schichten 14](#_Toc533063357)

[3. Quellen 15](#_Toc533063358)

1. Projektaufgabe
   1. Die 3 Schichtenarchitektur

Die 3 Schichtenarchitektur ist ein Modell wobei man seine Software in 3 Kategorien aufteilt (siehe Abbildung unten).



Benutzungsoberfläche(GUI-Schicht):

In dieser Schicht werden alle Steuerelemente und grafischen Komponenten platziert. In dieser Ebene der 3 Schichtenarchitektur werden jedoch keine logischen Prozesse ausgeführt hier wird lediglich ein Funktionsaufruf gemacht der dann die Daten an die Fachkonzeptschicht weiterleitet zur Verarbeitung

Fachkonzeptschicht:

In dieser Ebene werden jegliche Operationen und Verarbeitungen getätigt die von der Benutzungsoberfläche gestartet wurden. Wenn jedoch dabei Daten aus der Datenbank gebraucht werden oder aber Daten in die Datenbank geschrieben werden müssen, muss dafür die Datenhaltungsschicht angesprochen werden.

Datenhaltungsschicht:

In der letzten Ebene von der 3 Schichtenarchitektur handelt es sich um die letzte Schicht die dazu da ist Lese- und Schreibzugriffe auf der Datenbank zu verarbeiten um entweder Eingaben aus der Benutzungsoberfläche zu speichern oder aber Daten der Fachkonzeptschicht bereit zu stellen.

* 1. Das Design-Pattern: MVVM

Das für unserer Ansicht nach beste Design Pattern für User Interfaces ist MVVM.

Es steht für „Model – View –ViewModel“ und bietet einige Vorteile. Zu aller erst, sollte man aber klären was das MVVM ist. Wie bereits erwähnt steht es für Model View Viewmodel und ist somit in 3 Schichten eingeteilt.

Model:

Das Model sollte bekannt sein aus der normalen C# Programmierung. Es handelt sich hierbei um eine POCO Klasse (Plain Old Class Object), die keinen anderen Nutzen hat, als ein Objekt zu definieren. Als kleines Beispiel könnte man ein Objekt vom Typ Haus nehmen. Die POCO Klasse (also das Model) wäre hier HAUS. Hier wird beschrieben WAS jedes Objekt des Typs HAUS haben soll. Zum Beispiel ob es Fenster hat, Wie viele Eingänge das Haus hat, usw.…Die Besonderheit hierbei liegt darin, dass die POCO Klassen keine Methoden Enthalten. Sie definieren lediglich Wie etwas Aussieht, so dass es genutzt werden kann als Art Blaupause. Als Code würde unser Beispiel wie folgt aussehen:

public class Haus

{

public bool HatFenster { get; set; }

public int AnzahlEingänge { get; set; }

}

View:

Der View ist das, was der Anwender sieht. Genauer gesagt ist ein View das User Interface. Wir beziehen uns hierbei primär auf XAML der Markup Language. Hier wird der Code geschrieben, der die Steuerelemente hinzufügt. In diesem Dokument kann man mithilfe von Tags ein User Interface zusammenbauen. Ein Beispiel für das Steuerelement Button wäre wie folgt: <Button/>. In dieser Datei designend man auch die Steuerelemente. Zum Beispiel könnten wir unserem Button eine Höhe und eine Breite geben. Dazu noch den Inhalt was im Button stehen soll und schon ist er ganz ansehnlich: <Button Width=“60“ Height=“30“ Content=“Klick mich“/>.

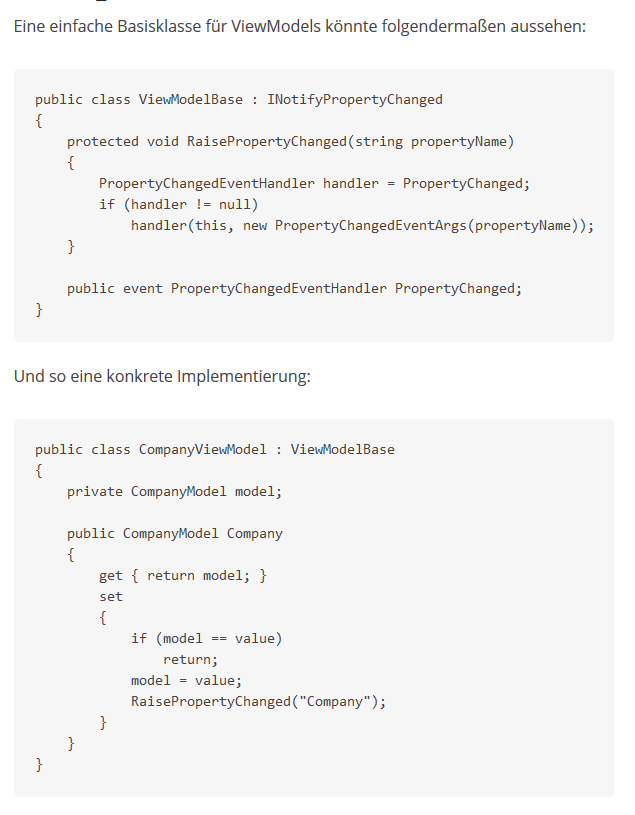
Auch die View ist Bekannt aus C# Projekten wie WPF (Windows Presentation Foundation) oder Windows Forms. Der spannende teil kommt aber erst jetzt. Das Viewmodel.

ViewModel:

Das ViewModel ist das, was dieses Design Pattern so besonders macht. Der Sinn des ViewModels ist, das Model und die dazugehörige Logik voneinander zu trennen bzw. sie Unabhängig voneinander zu machen. Wie wird das erreicht? Ich will nicht direkt darauf eingehen, wie das in C# umgesetzt wird. Dafür ist diese Beschreibung nicht gedacht. Es soll lediglich darüber aufklären wie das Konzept funktioniert.

Alles in allem kann man sich das so vorstellen: Die Daten des Models werden an der View mittels Datenbindung übergeben. Dafür Notwendig ist die Change Notification (in C# wäre es das Interface INotifyPropertyChanged), welche verlangt im Setter der benötigten Property das Interface zu implementieren. Somit wird jede Änderung des Models an den View übergeben.

Nichtsdestotrotz darf man das ViewModel nicht mit der Code Behind Datei verwechseln. Wichtig ist, dass man keine Abhängigkeit zu der View herstellt. Konkret gesagt: es ist nicht vorgesehen auf Elemente im View zuzugreifen, da dies die Unabhängigkeit zu Nichte machen würde. Es gibt ein Code Beispiel aus dem Internet, welches ich hier gerne verwenden würde:



Welche Vorteile bietet MVVM:

* Unabhängigkeit von User Interface und Business Logik
* Verschiedene Views können auf dasselbe View Model zugreifen
* Testbarkeit des View Models

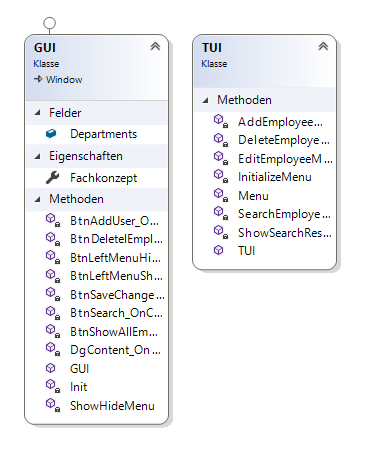
Es gibt aber auch Nachteile:

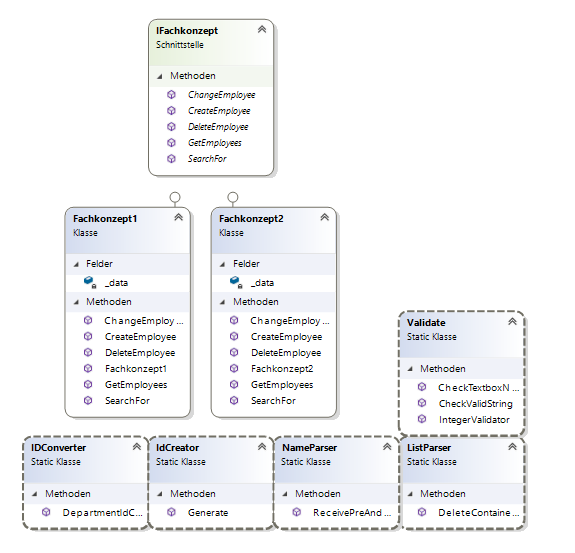
* Es kann zu einem höheren Rechenaufwand kommen was sich bei kleinen Maschinen auf die Usability auswirken kann
* MVVM ist bei kleineren UIs ein Overkill durch die Komplexität
  1. Unser Anwendungsproblem

Die Personalabteilung soll mit einer Software unterstützt werden, welches Eintrittsdatum die Mitarbeiter haben und welchen Abteilungen sie zugeordnet sind.

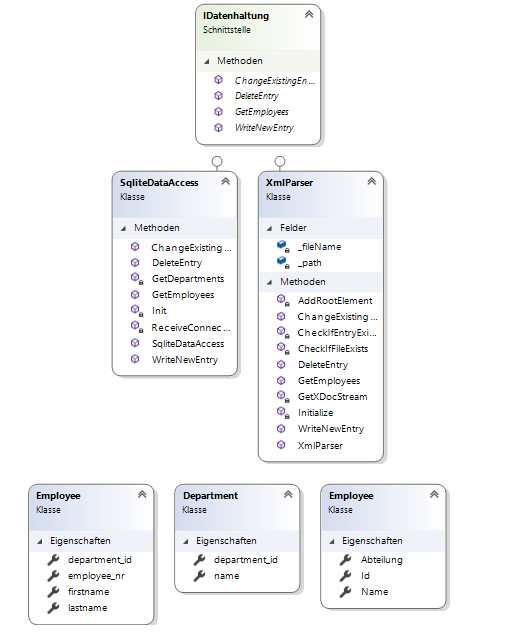
1. Das Produkt
   1. Klassendiagramm inkl. Zuordnung

Unser Projekt ist auf die 3 Schichten Architektur aufgebaut. Die Benutzungsoberfläche sieht in unserem Projekt wie folgt aus (Dargestellt als Klassendiagramm erzeugt von Visual Studio):

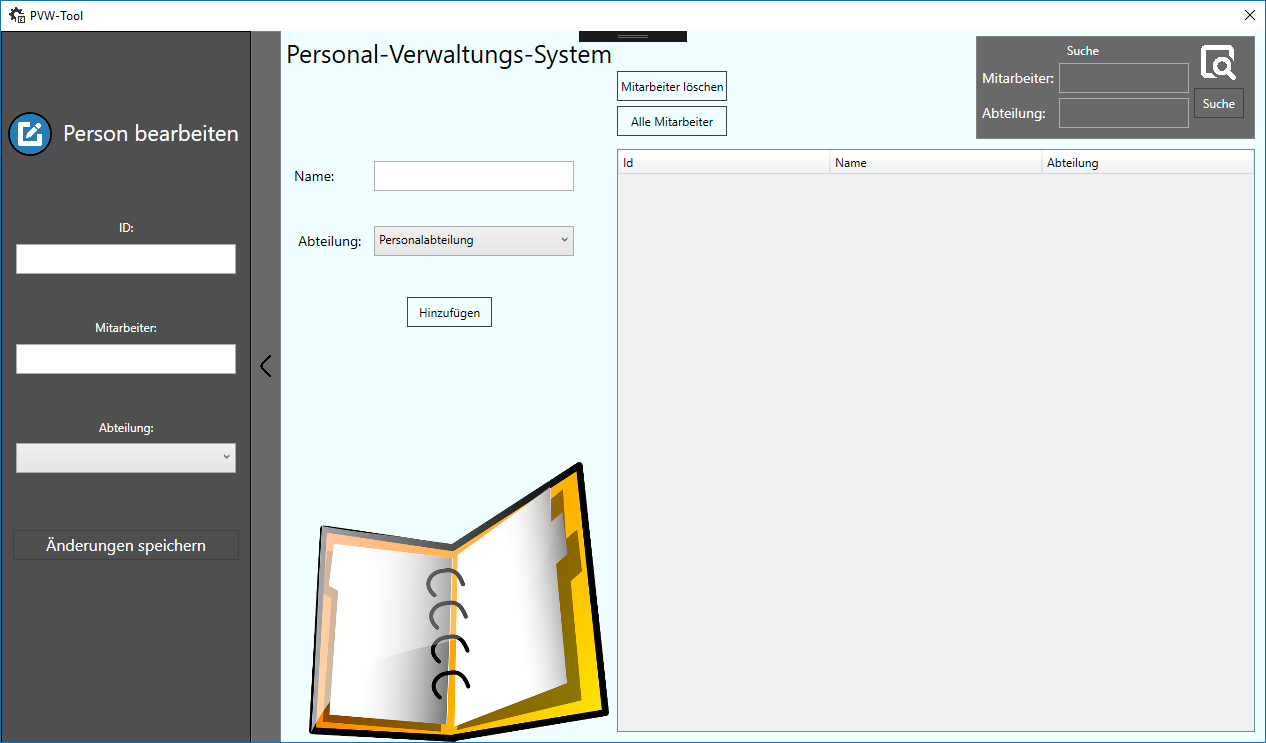


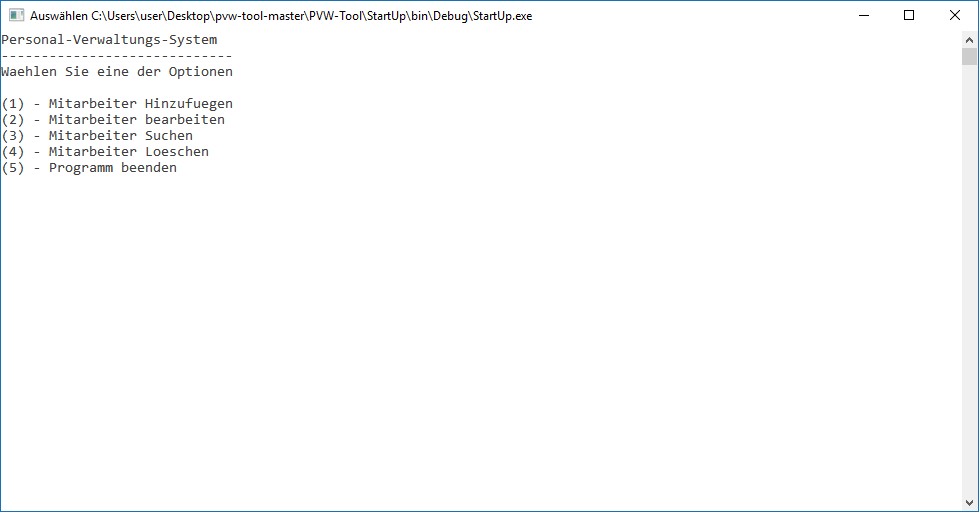
Die Fachkonzeptschicht besteht grundsätzlich nur aus Fachkonzept 1, Fachkonzept 2 und dem Interface IFachkonzept. Zudem habe wir aber noch selbstgeschriebene Extensionclasses, die uns helfen in der Validierung oder dem Parsen von Texten:

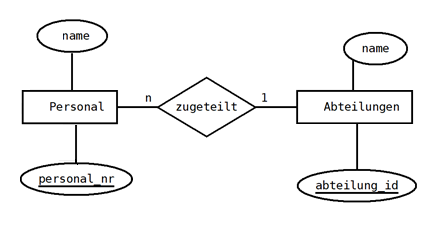
Unsere Datenhaltung (bestehend aus XML und SQLite) sieht im Klassendiagramm wie folgt aus:



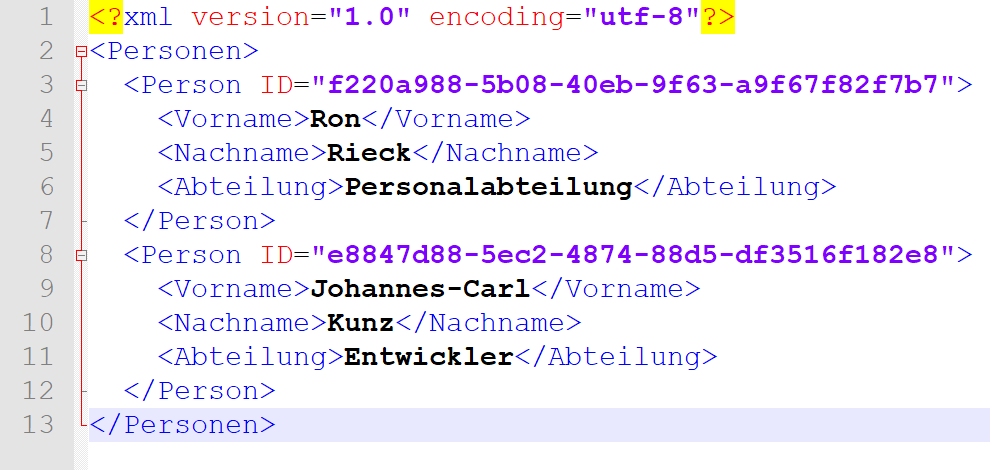
* 1. Benutzeroberfläche
     1. GUI



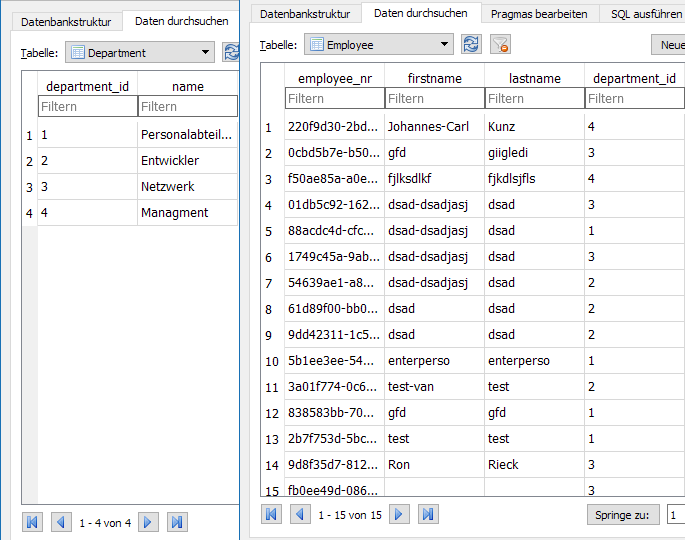
* + 1. TUI
  1. Datenhaltung
     1. ER-Modell



* + 1. XML



* + 1. SQLite

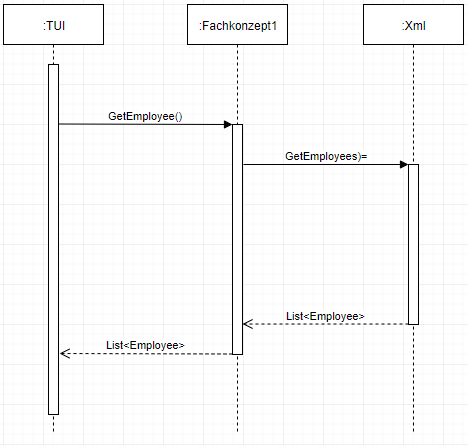


* 1. Realisierung des Designpatterns

Hat nicht stattgefunden durch Ausfall einer Person und Zeit gründen.

* 1. Sequenzdiagramm für die Kommunikation der 3 Schichten

In der folgenden Abbildung sehen sie ein Sequenzdiagramm was als Beispiel dient wie die Drei Schichten mit einander kommunizieren um eine Liste aller Mitarbeiter auszugeben.



1. Quellen

<https://www.norberteder.com/mvvm-das-viewmodel/>

<https://akrick.wordpress.com/2013/05/18/mvvm/>

<https://www.norberteder.com/mvvm-das-viewmodel/>

<https://blogs.msdn.microsoft.com/johngossman/2006/03/04/advantages-and-disadvantages-of-m-v-vm/>

<https://www.wintellect.com/model-view-viewmodel-mvvm-explained/>